Analyzing Software Errors in Safety-Critical, Embedded Systems

Robyn R. Lutz*
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

February 24, 1994

Abstract

This paper analyzes the root causes of safety-related software faults in safety-critical, embedded systems. The results show that software faults identified as potentially hazardous to the system are distributed somewhat differently over the set of possible error causes than non-safety-related software faults. Safety-related software errors are shown to arise most commonly from (1) discrepancies between the documented requirements specifications and the requirements needed for correct functioning of the system and (2) misunderstandings of the software's interface with the rest of the system. The paper uses these results to guide the identification of strategies to prevent such errors in other similar systems. The goal is to reduce safety-related software errors and to enhance the safety of complex, embedded systems.

1. Introduction

This paper examines 387 software faults uncovered during integration and system testing of two spacecraft, Voyager and Galileo. The standard IEEE definitions of a *fault* as "a manifestation of an error in software . . . . Synonymous with *bug*;" of an *error* as "human action that results in software containing a fault;" and of a *failure* as "an event in which a system or system component does not perform a required function within specified limits" are used here [7, 8]. Each of the 387 software faults was documented at the time of discovery by a form describing the anomaly or failure that indicated the existence of a software fault. The form also recorded the subsequent analysis and the corrective actions taken.

As part of the standard procedure for correcting each reported software fault, the failure effect of each is classified as negligible, significant, or catastrophic. Those classified as significant or catastrophic are investigated by a systems safety analyst as representing potential

*Author's mailing address is Dept. of Computer Science, Iowa State University, Ames, IA 50011. The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with NASA.

safety hazards [1 5], For this study the 74 (of 134) software faults on Voyager and 121 (of 253) software faults on Galileo documented as having potentially significant or catastrophic effects are classified as safety-related.

The spacecrafts' software is safety-critical in that it monitors and controls components that can be involved in hazardous system behavior [1 3]. The software must execute in a system context without contributing unacceptable risk.

Each spacecraft involves embedded software distributed on several different flight computers. Voyager has roughly 18,000 lines of source code; Galileo has over 22,000 [20]. Embedded software is software that runs on a computer system that is integral to a larger system whose primary purpose is not computational [7]. The software onboard the spacecraft controls the engineering and science acquisition) processes required for interplanetary missions. The software on both spacecraft is highly interactive in terms of the degree of message-passing among system components, the need to respond in real-time to monitoring of the hardware and environment, and the complex timing issues among parts of the system. The software development for each spacecraft involved multiple teams working for a period of years.

The purpose of this paper is to identify the extent and ways in which the cause/effect relationships of safety-related software errors differ from the cause/effect relationships of non-safety-related software errors. Preliminary results were reported in [1 6]. In particular, the analysis shows that human errors in identifying or understanding functional and interface requirements frequently lead to safety-related software faults. This distinction is used to guide the identification of error mechanisms through which the common human and process causes of the safety-related software faults studied here can be targeted during development. The goal is to improve system safety by understanding and, where possible, removing the prevalent sources of safety-related software errors.

The paper is organized as follows. Section II describes the methodology used. Section III presents the results of the analysis. Section IV indicates how these results fit into the context of prior work on software errors. Section V discusses some possible strategies for reducing safety-related software errors using the current results. Section VI provides a summary and identifies future work.

II. Methodology

A. Overview

The study described here characterizes the root causes of the safety-related software faults discovered during integration and system testing. The recent work by Nakajo and Kume on software error cause/effect relationships offers an appropriate framework for classifying the software errors [1 8]. Their work is extended here to account for the additional complexities operative in large, safety-critical, embedded systems with evolving requirements driven by hardware and environmental issues.

Previous studies of software errors have dealt primarily with fairly simple, non-embedded systems in familiar application domains (see Section IV for a discussion). Requirements specifications in these studies generally have been assumed to be correct, and safety issues have not been distinguished from program correctness. The work presented here instead

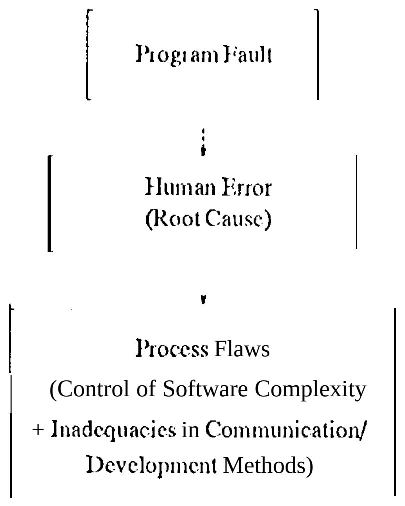

Figure 1: Analyzing Cause/Effect Relationships for Safety-Related Software Errors

builds on that in [18] to analyze software errors in safety-critical, embedded systems with evolving requirements.

Nakajo and Kume's classification scheme analyzes three points in the path from a software defect backwards to its sources. This approach allows classification not only of the documented program fault (the manifestation of an error in the software, e.g., an incorrect interface), but also of the earlier human error (the root c.dust, e.g., a misunderstanding of an interface specification), and of the even-earlier process flaws that contribute to the likelihood of the error's occurrence (e.g., inadequate communication between systems engineering and software development teams). Figure 1 presents a summary of these three points in the cause/effect analysis. The classification scheme thus leads backwards in time from the observed software fault to an analysis of the root cause (usually a communication error or an error in recognizing or deploying requirements), to an analysis of the software development process. An overview of the classification scheme, adjusted to the needs of safety-critics], embedded software, follows.

- Program Faults (Documented Software Errors)

  A. Internal Faults (e.g., syntax)

  B. Interface Faults (interactions with other system components, such as transfer of data or control)

  C. Functional Faults (operating faults: omission or unnecessary operations; conditional faults: incorrect condition or limit values; behavioral faults: incorrect behavior, not conforming to requirements)

- Human Errors (Root Causes)

  A. Coding or Editing Errors

B1. Communication Errors Within a Team (misunderstanding software interface specifications)

B2. Communication Errors Between Teams (misunderstanding hardware interface specifications or other team's software specifications)

C1. Errors in Recognizing Requirements (misunderstanding specifications or problem domain)

C2. Errors in Deploying Requirements (problems implementing or translating requirements into a design)

- Process Flaws (Flaws in Control of System Complexity + Inadequacies in Communication or Development Methods)

A. Inadequate Code Inspection and Testing Methods

B1. Inadequate Interface Specifications + Inadequate Communication (among software developers)

B2. Inadequate Interface Specifications + Inadequate Communication (between software and hardware developers)

C1. Requirements Not Identified or Understood + Incomplete Documentation

C2. Requirements Not Identified or Understood + Inadequate Design

By comparing common error mechanisms for the software faults identified as potentially hazardous with those of the other software faults, the prevalent root causes of the safety-related program faults are isolated, The classification of the sources of error is then applied here to determine countermeasures which may prevent similar error occurrences in other safety-critical, embedded systems. This paper thus uses the classification scheme to assemble an error profile of safety-related software errors and to identify development methods by which these sources of error may be able to be controlled in similar systems.

B. Classification Critera

Each program fault was classified and error causes assigned based on the information contained in the standard reporting form that documents each fault found during integration and system testing. This one-page form includes three textual descriptions which served as the primary source for the classification of the fault. The first description is of the observed problem or failure, written by the individual who observed it during integration or system testing. The second description is a later analysis of the error by the individual responsible for the module or component in which the problem occurred. This analysis may also expand or clarify the initial description of the problem by the originator. The third part of the form describes the corrective action taken to fix the problem (e. g., a software and documentation change). It also describes the test result, inspection, or review that confirms the adequacy of the correction to prevent recurrence. There are also several check-off boxes on the form, but these refer primarily to hardware issues (e.g., vibration testing, piece-part failure),

Additional pages (analysis results, test data, related memos) are sometimes attached to the form during the process of analysis and correction. This additional information was valuable in the study, as it provided insight into the human errors and process weaknesses that the spacecraft engineers and programmers saw.

in general, the primary criteria for classification was the documented judgment of the individuals who analyzed the error and validated that the required change in fact prevented the recurrence of the anomaly. Occasionally, overlapping faults or errors were documented on a single form (e.g. both an interface fault and a functional fault). in those cases, the classification reflects the programmer or engineer's judgment as documented on the form concerning which was the key cause of the observed problem. Additional discussion of error categorization is presented in [18] . An ongoing, multi-project investigation will address the issue of repeatability (do different analysts classify a given error in the same way?).

Clearly, the attribution of a key human error and. a key process flaw to each software fault oversimplifies the cause/effect relationship. However, the identification of these factors allows the characterization of safety-related software errors in a way that relates features of the development process and of the system under development to the safety consequences of those features. Similarly the association)) of each software fault with a human error, while unrealistic (in what sense is a failure to predict details of system behavior an error?), allows a. useful association between human factors (such as misunderstanding the requirements or the underlying physical realities) and their safety-related consequences.

III. Analysis of Safety -Related Software Defects

A. Program Faults

The six tables in the Appendix show the proportion and number (in parentheses) of non-safety-related software faults, errors, and process flaws as compared to safety-related software faults, errors, and process flaws for each of the two spacecraft. The results are summarized in the text in a series of bar graphs, The bar graphs contrast the distribution of error causes for all the safety-related faults with the distribution of error causes for all the non-safety-related faults. Any significant differences between the data from the two systems is discussed in the text with reference to the detailed tables in the Appendix.

Safety-related software faults account for about half of the total software faults discovered during integration and system testing on each of the two systems studied (55% for Voyager, 48% for Galileo), Few *internal* faults (e.g., coding errors internal to a software module) were uncovered during integration and system testing. An examination of software faults found later during operations also shows few internal faults. It appears that these coding errors are being detected and corrected before system testing begins. They thus are outside the scope of this paper and are not discussed further here.

The distribution of program faults for safety-related and non-safety-related faults in the two systems is shown in Figure 2. As can be seen, the two distributions display similar proportions of internal, interface, and functional faults. Functional faults (operating, conditional, or behavioral discrepancies from the functional requirements) account for almost three-quarters of both safety-related and non-safety-related program faults. The analysis summarized in Figure 2 and detailed in Table 1 of the Appendix also identifies *interface faults* (incorrect interactions with other system components, such as the timing or transfer

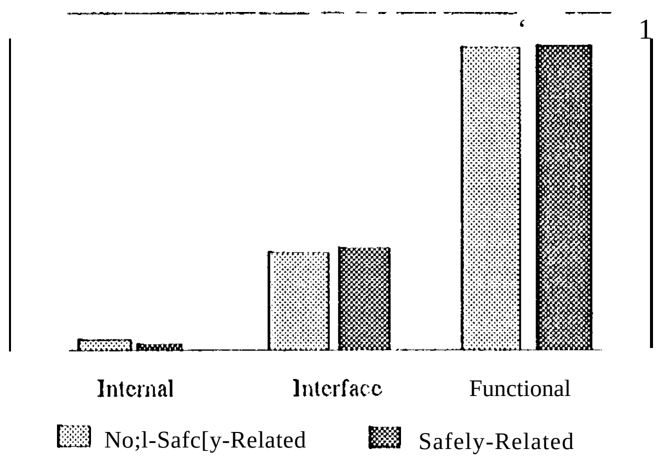

Figure 2: Distribution of Program Faults: Safety-Related and Non-Safety-Related

of data or control) as a significant problem (35% of the safety-related program faults on Voyager; 19% on Galileo). The high incidence of interface faults in these complex, embedded systems contrasts with the low incidence of interface faults in earlier studies on simpler, standalone software. (See Section IV for a discussion of this issue.)

Figure 3 examines the predominant type of program fault, the functional fault, in more detail, A significant difference between safety- related and non-safety-related conditional faults is apparent. Conditional faults (nearly always erroneous values on conditions or limits) tend to be safety-related in the two systems studied. Even though adjusting the values of limit variables during testing is considered to be fairly routine, the case of change obscures the difficulty of determining the appropriate value and the safety-related consequences of an inappropriate limit value. Erroneous values (e. g., of deadbands or delay timers) often involve risk to the spacecraft by causing inappropriate triggering of an error-recovery response or by failing to trigger a needed response. The association between conditional faults and safety - related software errors emphasizes the importance of specifying the correct, values for any data used in control decisions in safety-critical, embedded software.

Some differences between the distribution of faults on the two spacecraft exist (see 'Table 2). On Voyager fully half the safety-related functional faults are attributable to behavioral faults (the software behaving incorrectly). On Galileo, a slightly greater percentage is due to operating faults (nearly always a required but omitted operation in the software) than to behavioral faults. often the omitted operation involves the failure to perform adequate reasonableness checks on data input to a module. This frequently results in an error-recovery routine being called inappropriately.

B. Relationships Between Program Faults and Human Errors

Having classified each program fault, the second step in the cause/effect analysis is to trace backwards in time to the human factors involved in the program faults that were discovered

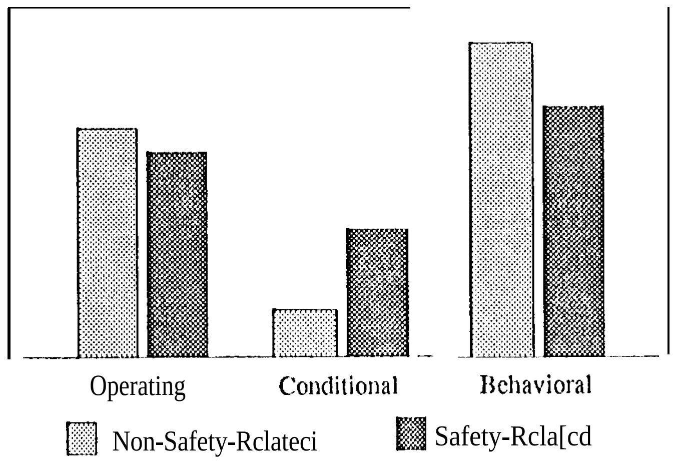

Figure 3: Distribution of Functional Faults

during integration and system testing. Figure 4 and Figure 5 summarize the relationships between the major types of program faults and the most frequent contributing errors for the two systems studied. For *interface faults,* the major human factors are either communication errors within a development team or communication errors between a development team and other teams. In the latter case, a further distinction is made between misunderstanding harclwarc/software interface specifications and misunderstanding the interface specifications with other software components.

Significant differences appear in the distribution) of error causes bet ween safet y-rel at ccl and non-safety-rc]atcd interface faults. Figure 4 shows that the primary human error causing safety-related *interface* faults is *misunderstood hardware/software interface specifications* (65% on Voyager; 48% on Galileo). Examples are faults caused by wrong assumptions about the initial state of relays or by unexpected heartbeat timing patterns in a particular operating mode. On the other hand, the human errors causing non-safety-mlatcd interface faults are distributed more evenly among the three categories. The profiles of safety-related interface errors assembled in Figure 4 and ‘J’able 3 of the Appendix emphasize the importance of developers understanding the software as a set of embedded components in a larger system.

The distribution of error causes for safety-related *functional faults* also differs substantially from the distribution of error causes for non- safety-related functional faults. Figure 5 identifies the primary cause of safety-rc]atcd *junctional faults* as errors in *recognizing (understanding) the requirements* (62% on Voyager, 79% on Galileo). On the other hand, non-safety-related fun ct ion al faults are more often cau sed by errors in deploying (implementing) the requirements.

q’able 4 provides further detail. Safety-related *conditional* faults (erroneous condition or limit values) are almost always caused by errors in *recognizing requirements.* Errors in recognizing requirements also cause safety-related *operational* faults (usually the omission of a required operation) and *behavioral* faults more often than errors in deploying requirements. g’able 4 reflects deficiencies in the documented requirements as well as instances of unknown (at the time of requirements specification) but necessary requirements for the two spacecraft,

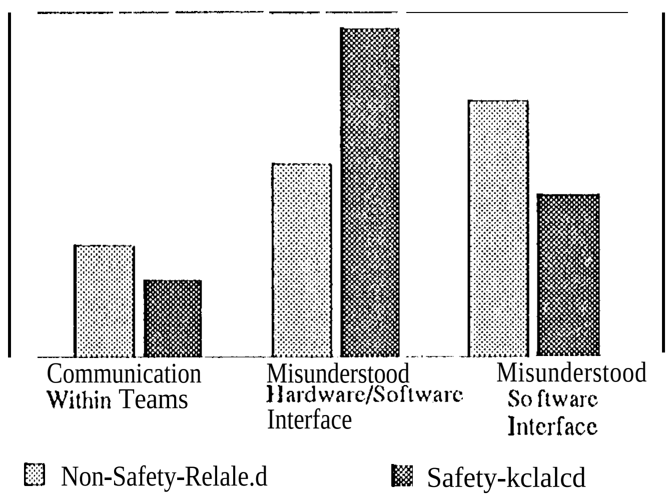

Figure 4: Root Causes (Human Errors) of Interface Faults

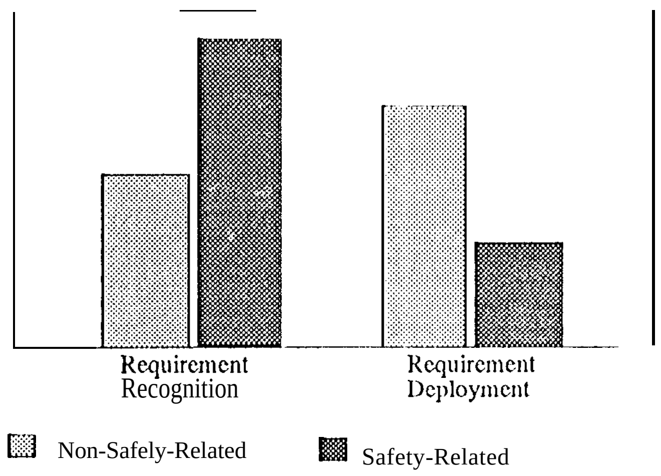

Figure 5: Root Causes (Human Errors) of Functional Faults

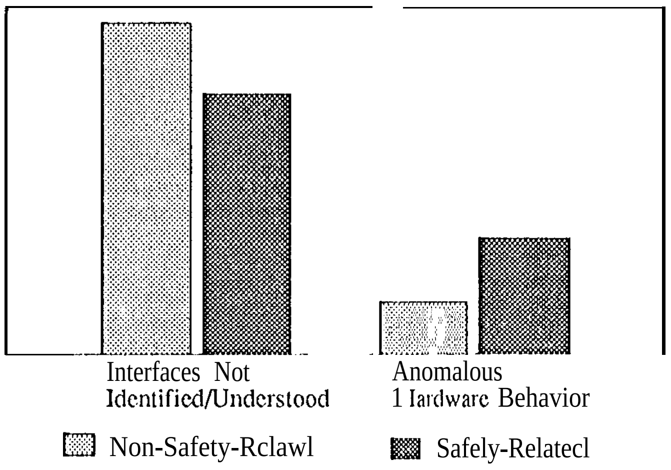

Figure 6: Process Flaws (Control of System Complexity) Causing Interface Faults

In summary, difficulties with requirements is the most common human error causing the safety-related software errors which have persisted until integration and system testing in these two systems. The tables point to errors in understanding the requirements specifications for the software/system interfaces as the most frequent cause of safety-related interface faults. Similarly, errors in recognizing the requirements is the most frequent cause leading to safety-related functional faults.

C. Relationships Between Human Errors and Process Flaws

In tracing backwards from the program faults to their sources, features of the system-development process can be identified which facilitate or enable the occurrence of errors. Discrepancies between the difficulty of the problem and the means used to solve it may permit hazardous software errors to occur [5].

The third step of the cause/effect analysis therefore associates a pair of process flaws with each program fault [1 8]. The first element in the pair identifies a process flaw or inadequacy in the *control of the system complexity (e.g.,* requirements which are not discovered until system testing). The second element of the pair identifies an associated process flaw in the *communication or development methods used* (e.g., imprecise or unsystematic specification methods).

The two elements of the process-flaw pair are closely related. Frequently, as is discussed in Sect. V, a solution to one flaw will provide a solution to the related flaw. For example, the lack of standardization evidenced by an ambiguous interface specification (an inadequacy in the control of system complexity) and the gap in interteam communication evidenced by a misunderstood interface specification (an inadequacy in the communication methods used) might both be addressed by the project-wide adoption of the same CASE tool.

Figure 6 relates interface faults and their human causes to process flaws involving system complexity. The figure shows that the process flaw *interfaces inadequately identified or understood,* a flaw which involves control of system complexity, is often associated with

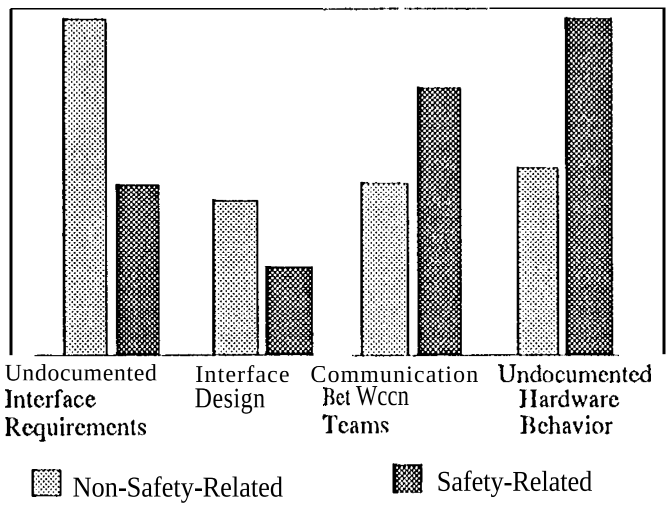

Figure 7: Process Flaws (Communication/Development Methods) Causing Interface Faults

both safety-related and non-safety-related interface faults. However, safety-related and non-safety-related interface faults differ in that *anomalous hardware behavior is* a more significant factor in safety-related than in non-safety-related interface faults. It is often associated with interface design during system testing, another indication of a unstable software product.

Figure 7 relates interface faults to process flaws involving communication or development methods. Again, differences in the distribution of causes for safety-related and non- safety-related interface errors is evident. *Undocumented interface requirements* is the most frequent flaw for non-safety-related interface faults, while *undocumented hardware behavior is the* most frequent flaw for safety-related interface faults.

Tables 5 and 6 show that there are significant differences in the process flaws that cause errors between the two spacecraft. Interface design during testing is involved in almost one-fifth of the safety-critical interface faults on Voyager, but in none of them on Galileo. This is because on Voyager a set of related hardware problems generated nearly half the safety-related interface faults. On the other hand, the problem of interface specifications that are known but not documented is more common on Galileo. This is perhaps due to the increased complexity of the Galileo interfaces. With regard to functional faults, although missing requirements are a frequent process flaw on both spacecraft, on Voyager inadequate design also occurs frequently, while on Galileo imprecise specifications often occur.

Figure 8 summarizes the relationships between process flaws involving control of system complexity and functional faults. For *functional faults*, requirements not identified and requirements not understood are the most common complexity-control flaws. Safety-related functional faults are more likely than non-safety-related functional faults to be caused by *requirements which have not been identified* (55% *vs.* 41 %).

With regard to process flaws involving communication or development methods, Figure 9 shows that *missing requirements* are involved in nearly half (42%) of the safety-related functional faults, but in only 25% of the non-safety-related functional faults, *Imprecise or*

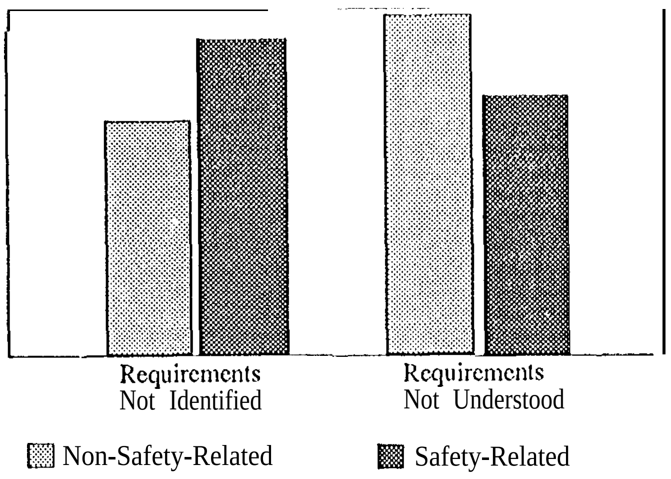

Figure 8: Process Flaws (Control of System Complexity) Causing Functional Faults

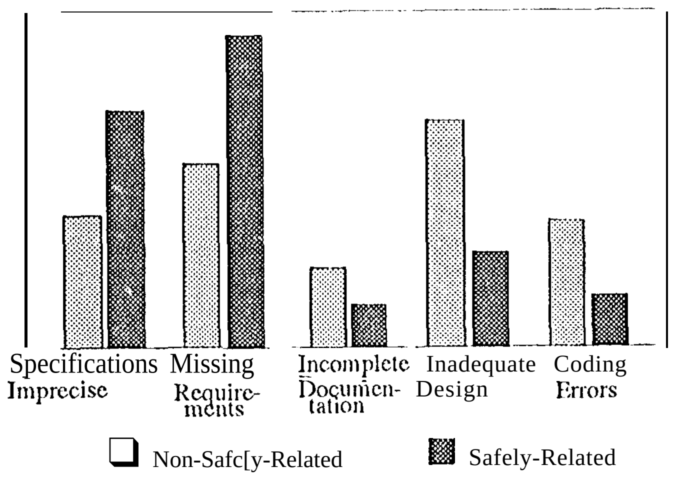

Figure 9: Process Flaws (Communication/Development Methods) Causing Functional Faults

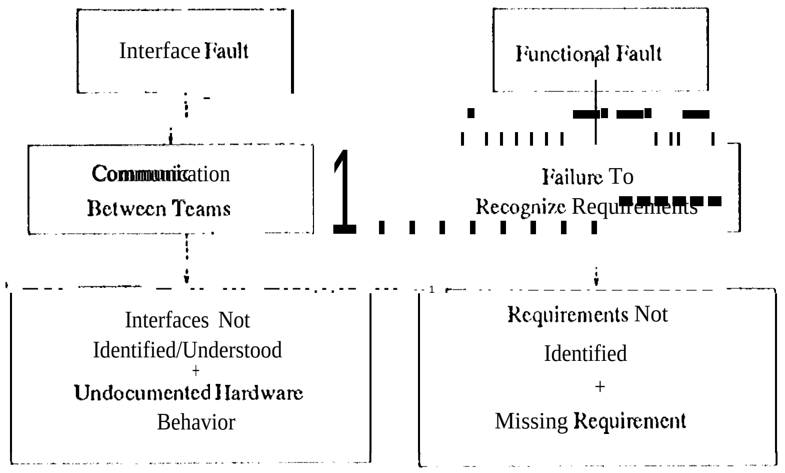
Interface Fault
Functional Fault
Communication Between Teams
Failure To Recognize Requirements
Interfaces Not Identified/Understood + Undocumented Hardware Behavior
Requirements Not Identified + Missing Requirement

Figure 10: Frequent Cause/Effect Relationships for Safety-Related Software Errors

unsystematic specifications arc also more often a factor in safety-related than in non-safety - related functional faults. These results suggest that the sources of safety-rclatccl software errors lie farther back in the software development process-in inadequate specification or understanding of requirements-whereas the sources of non-safety-related errors more commonly involve inadequacies in the design phase.

Figure 10 summarizes these results graphically. It displays two of the cause/effect mechanisms that occur frequently in the two spacecraft, resulting in safety-related interface faults an c1 safet y-related functional faults.

IV. Comparison of Results with Previous Work

Although software errors and their causes have been studied extensively, the current work differs from most of the prior investigations in the following four ways:
1) The software chosen for analysis in most studies is not embedded in a complex system as it is here. The consequence is that the role of interface specifications in controlling software hazards has been underestimated.
2) Unlike the current paper, most studies have analyzed fairly simple systems in familiar and wcl]-understood application domains. Consequently, fcw software faults have been found during system testing in most studies, leading to a gap in knowledge regarding the sources of these more-persistent and often more hazardous faults.
3) Most studies assume that the requirements specification is correct. On the spacecraft, as in many large, complex systems, the requirements evolve as knowledge of the system's behavior and the problem domain evolve. Similarly, most studies assume that requirements arc fixed by the time that systems testing begins. This leads to a underestimation of the impact of unknown requirements on the scope and schedule of the later stages of the software development process.
4) The distinction between causes of safety-critical ancl non-safety-critical software errors has

not been adequately investigated. Efforts to enhance system safety by specifically targeting the causes of safety-related errors, as distinguished from the causes of all errors, can take advantage of the distinct error mechanisms, as described in Sect. 5.

A brief description of the scope and results of some related work is given below and compared with the results presented in this paper for safety-critical, embedded computer systems.

Nakajo and Kume categorized 670 errors found during the software development of two firmware products for controlling measuring instruments and two software products for instrument measurement programs [18]. Over *90%* of the errors were either interface or functional faults, similar to the results reported here.

Unlike the results described here, Nakajo and Kume found many conditional faults. It may be that unit testing, as on the spacecraft) finds many of the conditional faults prior to system testing. While the key human error on the spacecraft involved communication between teams, the key human error in their study involved communication within a development team. Both studies identified complexity and documentation deficiencies as issues. However, the software errors on the spacecraft tended to involve inherent technical complexity, while the errors identified in the earlier study involved complex correspondences between requirements and their implementation. Finally, the key process flaw that they identified was a lack of methods to record known interfaces and describe known functions. In the safety-c.ritical, embedded software on the spacecraft, the flaw was more often a failure to identify or to understand the requirements.

Ostrand and Weyuker categorized 173 errors found during the development and testing of an editor system [21]. Only 2% of the errors were found during system testing, reflecting the simplicity and stability of the interfaces and requirements. Most of the errors (61%) were found instead during function testing. Over half these errors were caused by omissions, confirming the findings of the present study that omissions are a major cause of software errors,

Schneidewind and Hoffmann [24] categorized 173 errors found during the development of four small programs by a single programmer. Again, there were no significant interfaces with hardware and little system testing. The most frequent class of errors, other than coding and clerical, was design errors. All three of the most common design errors--extreme conditions neglected, forgotten cases or steps, and loop control errors- are also common functional faults on the spacecraft.

Both the findings presented in [21, 24] and in this paper confirm the common experience that early insertion and late discovery of software errors maximizes the time and effort that the correction takes. Errors inserted in the requirements and design phases take longer to find and correct than those inserted in later phases (because they tend to involve complex software structures). Errors discovered in the testing phase take longer to correct (because they tend to be more complicated and difficult to isolate). This is consistent with the results in [20] indicating that more severe errors take longer to discover than less severe errors during system-level testing. Furthermore, this effect was found to be more pronounced in more complex (as measured by lines of code) software.

The work done by Endres is a direct forerunner of Nakajo and Kume's in that Endres backtracked from the error type to the technical and organizational causes which led to each type of error [5], Moreover, because he studied the system testing of an operating system, the software's interaction with the hardware was a source of concern. Endres noted the difficulty of precisely specifying functions] demands on the systems before the programmer had seen their effect on the dynamic behavior of the system. His conclusion that better tools were needed to attack this problem still holds true eighteen years after he published his study.

Of the 432 errors that Endres analyzed, 46% were errors in understanding or *communi*-cating the problem, or in the choice of a solution, 38% were errors in implementing a solution, and the remaining 16% were coding errors. These results are consistent with the finding here that software with many system interfaces displays a higher percentage of software errors involving understanding requirements or the system implications of alternative solutions.

Eckhardt et al., in a study of software redundancy, analyzed the errors in twenty independent versions of a software component of an inertial navigation system [4]. They found that inadequate understanding of the specifications or the underlying coordinate system was a major contributor to the program faults causing coincident failures,

Addy, looking at the types of errors that caused safety problems in a large, real-time control system, concluded that the design complexity inherent in such a system requires hidden interfaces which allow errors in non-critical software to affect safety-criticaJ software []]. This is consistent with Selby and Basili's results when they analyzed 770 software errors during the updating of a library tool [25]. Of the 46 errors documented in trouble reports, 70% were categorized as "wrong" and 28% as '(missing." They found that subsystems that were highly interactive with other subsystems had proportionately more errors than less interactive subsystems.

Chill arege et al., classified defects from several operating systems and database management systems according to type and trigger (e.g., boundary condition, exception handling, etc.) [2]. By comparing the distribution of defect types to the expected distribution for each stage of development, the progress of a product's d evelopment can be monitored. Based on the authors' experience with similar products, they expect to find function defects peaking at design testing, interface errors peaking at integration testing, and timing/serialization errors peaking at system testing for the systems they analyze. The prevalence of interface and timing errors is consistent with the data from the two spacecraft studied here. However, the spacecraft both continue to display functional defects throughout system testing. This is probably attributable to the continued evolution of software requirements driven by the hardware, the interplanetary environment, and unique mission science needs.

Leveson listed a set of common assumptions that are often false for control systems, resulting in software errors [13]. Among these assumptions arc that the software specification is correct, that it is possible to predict realistically the software's execution environment (e.g., the existence of transients), and that it is possible to anticipate and specify correctly the software's behavior under all possible circumstances. These assumptions tend to be true for the simple systems in which software errors have been analyzed to date and false for spacecraft and other large, safety-critical, embedded systems. '1'bus, while studies of software errors in simple systems can assist in understanding internal errors or some functional errors, they are of less help in understanding the causes of safety-related software errors, which tend heavily to involve interfaces or recognition of complex requirements.

Similarly, standard measures of the internal complexity of modules have limited usefulness in anticipating software errors during system testing. It is not the internal complexity of a module but the complexity of the module's connection to its environment that yields the persistent, safety-related errors seen in the embedded systems here [10].

V. Discussion

The results in Sect. III indicate that safety-related software defects are distributed somewhat differently over the set of possible causes than non-safety-related software defects in the systems studied. Finding the prevalent causes of safety-related software errors in these two systems may help guide the development of strategies to reduce such errors in other similar systems. By targeting the causes of safety-related errors, system safety may be directly enhanced.

The results of the analysis of safety-related software errors in the two spacecraft can be interpreted as guidelines for preventing such errors in future, similar systems.

1. *Focus on the interfaces between the software and the system in analyzing the problem domain, since these interfaces are a major source of safety-related software errors.*

The traditional goal of the requirements analysis phase is the specification of the software's external interface to the user. This definition is inadequate when the software is deeply embedded in larger systems such as spacecraft, advanced aircraft, air-traffic control units, or manufacturing process-control facilities. In such systems, the software is often physically and logically distributed among various hardware components of the system. The hardware involved may be not only computers but also sensors, actuators, gyros, and science instruments [11].

Specifying the external behavior of the software (its transformation of software inputs into software outputs) only makes sense if the interfaces between the system inputs (e.g., environmental conditions, power transients) and the software inputs (e.g., monitor data) are also specified. Similarly, specifying the interfaces- especially the timing and dependency relationships- between the software outputs (e. g., star identification) and system outputs (e.g., closing the shutter on the star scanner) is necessary. [6, 12]

System-development issues such as timing (real-time activities, interrupt handling, frequency of sensor data), hardware capabilities and limitations (storage capacity, power transients, noise characteristics), communication links (buffer and interface formats), and the expected operating environment (temperature, pressure, radiation) need to be reflected in the software requirements specifications because they are frequently sources of errors involving interfaces.

Timing is a particularly difficult source of safety-related software interface errors since timing issues are so often integral to the functional correctness of safety-critical, embedded systems, Timing dependencies (e.g., how long input data is valid for making control decisions) should be included in the software interface specifications. Analytical models or simulations to understand system interfaces are particularly useful for complex, embedded systems.

2. *Identify safety-critical hazards early in the requirements analysis.*

These hazards are constraints on the possible designs and factors in any contemplated tradeoffs between safety (which tends to encourage software simplicity) and increased functionality (which tends to encourage software complexity) [12, 25]. Many of the safety-related soft ware errors in the two spacecraft involve dat a ob jects or processes that would be t argeted for special attention using hazarcl-detection techniques such as those described in [9, 13]. Early detection of these safety-critical objects and increased attention to the software operations that usc them might forestall associated safety-related software errors.

3. Use *formal specification techniques in addition to natural-language software requirements specifications.*

Lack of precision and incomplete requirements led to many of the safety-rc]atccl software errors seen here. Enough detail is needed to cover all circumstances that can be envisioned (component failures, timing constraint violations, expired data) as well as to document all environmental assumptions (e.g., how c1osc to the sun an instrument will point) and assumptions about other parts of the system (maximum transfer rate, consequences of race conditions or cycle slippage). The capability to describe dynamic events, the timing of process interactions in distinct computers, decentralized supervisory functions, etc., should be considered in chooosing a formal method [3, 6, 17, 22, 23, 26]. Since embedded software systems arc often quite large, formally specifying or analyzing the entire system may not be feasible. Data on causes of safety-related errors may help guide the selection of the portions of the software most likely to benefit from the added rigor of formal methods. In a spacecraft currentl y un dcr development, for example, error recovery software and critical int erfaces have been identified for cxperiments in formal specifi cation.

4. Promote informal communication among teams.

Many safety-related software errors resulted from onc individual or team misunderstanding a requirement or not knowing a fact about the system that member(s) of another development t cam knew. The goal is to modularize responsibility in a development project without modularizing communication about the system under development. The identification and tracking of safety hazards in the two systems described here, for example, is clearly best done across team boundaries,

5. As requirements evolve, communicate the changes to the development and test teams.

This is both more important (because there arc more requirements changes during design and testing) and more difficult (because of the number and size of the teams and the length of the development process) in a large, embedded system than in simpler systems. in analyzing the safety-related software errors, it is evident that the determination as to who needs to know about a change is often made incorrectly. Frequently, changes that appear to involve only onc team or system component end up affecting other teams or components at some later date (sometimes as the result of incompatible changes in distinct units).

There is also a need for faster distribution of changes that have been made, with the update stored so as to be fingertip accessible. CASE tools offer a possible solution to the difficulty of promulgating change without increasing paperwork.

The prevalence of safety-related software errors involving misunderstood or missing requirements points up the inadequacy of consistency checks of requirements and code as a means of demonstrating system correctness [1 2]. Code that implements incorrect requirements is incorrect if it fails to provide needed system behavior.

Similarly, generating test cases from misunderstood or missing requirements will not test system correctness. Traceability of requirements and automatic test generation from specifications offers only partial validation of complex, embedded systems. Alternative validation and testing methods such as those described in [1 1, 13] offer greater coverage.

6. *Include requirements for "defensive design"* [1 9].

Many of the safety-related software errors involve inadequate software responses to extreme conditions or extreme values. Anomalous hardware behavior, unanticipated states, events out of order, and obsolete data all contribute to safety-related software errors on the spacecraft.

Run-time safety checks on the validity of input data, watchdog timers, delay timers, software filters, software-imposed initialization conditions, additional exception handling, and assertion checking can be used to combat the many safety-critical software errors involving conditional and omission faults [13]. Requirements for error-handling, overflow protection, signal saturation limits, heartbeat and pulse frequency, maximum event duration, and system behavior under unexpected conditions can be added and traced into the design. Many safety-related functional faults involve error-recovery routines being invoked inappropriately because of erroneous limit values or bad data.

Backward analysis from critical failures to possible causes offers one check of how defensive the requirements and design are [14], Requirements specifications that account for worst-case scenarios, models that can predict the range of possible (rather than allowable) values, and simulations that can discover unexpected interactions before system testing contribute to the system's defense against hazards.

VI. Summary and Future Work

In large, embedded systems such as the two spacecraft in this study, the software requirements change throughout the software development process, even during system testing. This is largely due to unanticipated behavior, dynamic changes in the operating environment, and complex software/hardware and software/software interactions in the systems being developed. Controlling requirement changes (and, hence, the scope and cost of development) is difficult since the changes are often prompted by an improved understanding of the software's necessary interfaces with the physical components of the spacecraft in which it is embedded. Complex timing issues and hardware idiosyncrasies often prompt changes to requirements or to design solutions.

The analysis presented here of the cause/effect relationships of safety-related software errors pinpoints aspects of system complexity which merit additional attention, Specifically, the results have shown that conditional faults (e.g., condition or limit values) are highly correlated with safety-related software errors. Operating faults (especially the omission of run-time reasonableness checks on data) are also highly correlated with safety-related software errors. Unknown, undocumented, or erroneous requirements frequently are associated with safety-related software errors as well. Hardware/software interfaces have been shown to be a frequent trouble spot because of the lack of communication between teams.

The results presented in this paper indicate a need for better methods to confront the real-worlcl issues of developing safety-critical, embedded software in a complex, distributed system. Future work will be directed at incorporating knowledge of the distinct. error mechanisms that produce safety-related software errors into the requirements analysis and validation processes. Follow-on studies will evaluate the adequacy and repeatability of the error classifications used here. Additional experiments to test the proposed guidelines both in similar, future systems and in a cross-section of embedded software would be useful. Work is also needed on specifying how the results presented in this paper can be used to predict more precisely what features or combinations of factors in a safety-critical, embedded system are likely to cause time-consurning and hazardous software errors.

References

[1] E. A. Addy, "A Case Study on Isolation of Safety-Critical Software," in *Proc 6th Annual Conf on Computer Assurance.* NIST/IEEE, 1991, pp. 75-83.

[2] R. Chillarege, et al., "Orthogonal Defect Classification- A Concept for 111-1'recess Measurement s," *IEEE Trans Software Eng, 1S, 11, Nov* 1992, *pp. 943-956.*

[3] A. M. Davis, *Software Requirements, Analysis and Specification.* Englewood Cliffs, N. J.: Prentice Hall, 1990.

[4] D. E. Eckhardt, et al., "An Experimental Evaluation of Software Redundancy as a Strategy for improving Reliability," *IEEE Tram Software Eng, 17, 7,* July 1991, pp. *692-702.*

[5] A. Endres, "An Analysis of Errors and 'J'heir Causes in Systems Programs," *IEEE Trans Software Eng*, SE-I, *2,* June 1975, pp. 140-149.

[6] E. M. Gray and R. 11. Thayer, "Requirements," in *A erospace Software Engineering, A Collection of Concepts.* Ed. C. Anderson and M. Dorfman. Washington: AlAA, 1991, pp. 89--121.

[7] ANSI/IEEE Standard Glossary of Software Engineering Terminology. New York: IEEE, 1983.

[8] IEEE Standard Dictionary of Measures To Produce Reliable Software, Std 982,1-1988, Ncw York: IEEE, 1989.

[9] M. S. Jaffe et al., "Software Requirements Analysis for Real-Time Process- Control Systems," *IEEE Trans Software Eng, 17, 3,* March 1991, pp. *241-258,*

[10] 1'. Jalote, An *Integrated Approach to Software Engineering. New* York: Springer-Vcrlag, 1991.

[11] J. C. Knight, "Testing," in *Aerospace Software Engineering, A Collection of Concepts. Ed. C.* Anderson and M. Dorfman. Washington: AIA A, 1991, pp. 135--159.

[12] N. G. Leveson, "Safety," in *Aerospace Software Engineering, A Collection of Concepts.* Ed. *C.* Anderson and M. Dorfman. Washington: AIA A, 1991, pp. 319-336.

[13] N. G. Leveson, "Software Safety in Embedded Computer System s," *Commun A CM,* Vol. 34, No. 2, Feb 1991, pp. 35--46.

[14] N. G. Leveson and P. R. Harvey, "Analyzing Software Safety," *IEEE Transactions on Software Engineering,* SE-9, *5,* Sept *1983, pp. 569-579.*

[15] Karan L'Heureux, "Software Systems Safety Program RTOP, Phase A Report," Internal Document, Jet l'repulsion laboratory, April 19, 1991,

[16] R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proc IEEE Internat Symp on Requirements Engineering*. Los Alamitos, CA: IEEE Computer Society Press, 1993, pp. 126-133.

[17] R. Lutz aud J. S. K. Wong, "Detecting Unsafe Error Recovery Schedules," *IEEE Trans Software Eng*, 18, 8, Aug, 1992, pp. 749-760.

[18] T. Nakajo and 11. Kume, "A Case History Analysis of Software Error Cause -Effect Relationships," *IEEE Trans Software Eng 17, 8,* Aug 1991, *pp. 830-838,*

[19] P. G. Neumann, "The Computer-ltc]atcd Risk of the Year: Weak Links and Correlated Events," in *Proc 6th Annual c'onf on Computer Assurance.* NIST/IEEE, 1991, pp. 5-8.

[20] A, P. Nikora, "Error Discovery Rate by Severity Category and Time to Repair Software Failures for Three J] 'I, Flight Projects," Internal Document, Jet Propulsion laboratory, 1991.

[21] 'J'. J. Ostrand and E. J. Weyuker, "Collecting and Categorizing Software Error Data in an industrial Environment," *The Journal of Systems and Software*, 4, 1984, pp. 289-300.

[22] *Proc* Berkeley *Workshop on Temporal and Real-Time Specification. Eels. P.* B. Ladki n and F. H. Vogt. Berkeley, CA: International Computer Science Institute, 1990, TR-90-060.

[23] J. Rushby, "Formal Methods and Digital Systems Validation for Airborne Systems," CSL Technical Report, SRI-CSL-93-07, Nov 1993.

[24] N. F. Schneidewind and 11,-M. Hoffmann, "An Experiment in Software Error Data Collection and Analysis," *IEEE Trans Software Eng*, SE-5, 3, May 1979, pp. 276-286.

[25] R. W. Selby and V. R. Basili, "Analyzing Error-Prone System Structure," *IEEE Trans Soft-. ware Eng* 17, 2, Febr 1991, pp. 141-152.

[26] J. M. Wing, "A Specifier's Introduction to Formal Methods,)' Computer, Vol. 23, Sept 1990, pp. 8-26.

Appendix

	Non-Safety-Related Program Faults				*Safety-Related Program Faults*			
	Voyager	*(60)*	*Galileo*	*(132)*	*Voyager*	*(74)*	*Galileo*	*(121)*
Internal	2%	(1)	4%	(5)	0%	(o)	2%	(2)
Interface	33%	(20)	18%	(24)	35%	(26)	19%	(23)
Functional	65%	(39)	78%	(103)	65%	(48)	79%	(96)

Table 1: *Classification of Program Faults*

	Non-Safety-Related Functional Faults				Safety-Related Functional Faults			
	Voyager	(39)	Galileo	(103)	Voyager	(48)	Galileo	(96)
Operating	26%	(10)	43%	(44)	19%	(9)	43%	(41)
Conditional	20%	(8)	4%	(4)	31%	(15)	18%	(17)
Behavioral	54%	(21)	53%	(55)	50%	(24)	40%	(38)

Table 2: *Classification of Functional Faults*

	Non-Safety-Related Interface Faults				Safety-Related Interface Faults			
	Voyager	(20)	Galileo	(24)	Voyager	(26)	Galileo	(23)
I. Intra-team Communication	5%	(1)	33%	(8)	8%	(2)	22%	(5)
II. Interteam Communication:								
Hardware/Software Interface	30%	(6)	38%	(9)	65%	(17)	48%	(11)
Software Interfaces	65%	(13)	29%	(7)	27%	(7)	30%	(7)

Table 3: *Relationships of Root Causes (Human Errors) to Interface Faults*

	Non-Safety-Related Functional Faults				Safety-Related functional Faults			
	Voyager	(39)	Galileo	(103)	Voyager	(48)	Galileo	(96)
I. Requirement Recognition:								
Operating	10%	(4)	17%	(17)	8%	(4)	33%	(32)
Conditional	8%	(3)	0%	(1)	25%	(12)	16%	(15)
Behavioral	10%	(4)	29%	(30)	29%	(14)	30%	(29)
Total	28%	(11)	47%	(48)	62%	(30)	79%	(76)
II. Requirement Deployment:								
Operating	15%	(6)	26%	(27)	11%	(5)	9%	(9)
Conditional	13%	(5)	3%	(3)	6%	(3)	2%	(2)
Behavioral	44%	(17)	24%	(25)	21%	(10)	9%	(9)
Total	72%	(28)	53%	(55)	38%	(18)	21%	(20)

Table 4: *Relationships of Root Causes (Human Errors) to Functional Faults*

	Non-Safety-Related Interface Faults				*safety- Related Interface Faults*			
	Voyager		*Galileo*	*(24)*	*Voyager*		*Galileo*	
I. Control of System Complexity:								
interfaces not understood	90%	(18)	83%	(20)	54%	(14)	87%	(20)
Hardware anomalies	10%	(2)	17%	(4)	46%	(12)	13%	(3)
II. Communication/Development:								
Interface specifications	35%	(7)	42%	(10)	8%	(2)	35%	(8)
Interface design lags	35%	(7)	4%	(1)	19%	(5)	0%	(0)
Interteam communication	20%	(4)	21%	(5)	27%	(7)	35%	(8)
Undocumented hardware	10%	(2)	33%	(8	46%	(12)	30%	(7)

Table 5: *Process Flaws Causing Interface Faults*

	Non-Safety-Related Functional Faults				*Safety-Related Functional Faults*			
	Voyager		*Galileo*	*(103)*	*Voyager*	*(48)*	*Galileo*	
1. Control of System Complexity:								
Requirements not identified	28%	(11)	46%	(47)	44%	(21)	60%	(58)
Requirements not understood	72%	(28)	54%	(56)	56%	(27)	40%	(38)
II. Communication/Development Causing Errors in Requirements Recognition:								
Imprecise specification	10%	(4)	18%	(19)	21%	(10)	38%	(36)
Missing requirements	18%	(7)	28%	(29)	42%	(20)	42%	(40)
Communication/Development Causing Errors in Requirements Deployment:								
incomplete documentation	10%	(4)	12%	(12)	2%	(1)	8%	(8)
Persistent coding errors	28%	(11)	13%	(13)	10%	(5)	5%	(5)
Inadequate design	33%	(13)	29%	[30)	25%	(12)	7%	(7)

'1'able 6: *Process Flaws Causing Functional Faults*

Index Terms

Software errors, software safety, requirements analysis, embedded software, system testing, software specification, safety-critical systems, spacecraft,